MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE LABORATORY

A.I. MEMO 543

30 AUGUST 1979

PROCEDURAL ATTACHMENT

LUC STEELS

ABSTRACT

A frame-based reasoning system is extended to deal with procedural attachment. Arguments are given why procedural attachment is needed in a symbolic reasoner. The notion of an infinitary concept is introduced. Conventions for representing procedures and a control structure regulating their execution is discussed. Examples from electrical engineering and music illustrate arithmetic constraints and constraints over properties of strings and sequences.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

TABLE OF CONTENTS

1. INTRODUCTION

2. INFINITARY CONCEPTS

3. PROCEDURAL ATTACHMENT

3.1. REPRESENTING PROCEDURES AND VALUES

3.2. EVALUATION

4. EXAMPLES

4.1. ELECTRONIC CIRCUITS

4.2. MUSIC

5. CONCLUSION

6. REFERENCES

1. INTRODUCTION

We will study problems of procedural attachment within the context of an existing reasoning system. By procedural attachment we will mean that a procedure is used instead of a description to constrain a particular object in the model of a certain problem situation.

The reasoning system which we will use as basis for the discussion is the XPRT-system extensively discussed in Steels (1979). The XPRT-system (pronounced expert-system) is a package designed to serve as a foundation for constructing knowledge-based expert systems. It features a natural language like description language (called XPRTESE) and a reasoning component. For each of the concepts he wants to use, the user defines a *frame*. A frame contains aspects for each important question to be asked about the class of situations to which the concept applies. The frame also contains constraints in the form of natural language like descriptions which are partial answers or ways to find the answer to each question posed by a frame.

Given such a series of frames, the user then specifies a problem situation by (partially) describing it. Based on these initial specifications the reasoner will build up a model which contains the objects in the situation, the initial properties and relations, and all properties and relations which are deduced by antecedent-reasoning over the constraints attached to the instantiated frames or by goal-directed (consequent) reasoning based on so called method-frames associated with each concept. New specifications can be added interactively and questions (including requests for justification) can be asked.

So far the user of the XPRT-system does not have to introduce procedures at all. Every activity consists of a deductive operation such as decomposition, instantiation, distribution of descriptions, application of modus ponens, etc. Moreover the system is an open system in the sense that this system is adequate to deal with any domain and allows for sophisticated problem solving. All this contrasts of course with other frame-based knowledge representation efforts (such as FRL (Goldstein and Roberts ,1977)) where the amount of deductive support is minimal and the user is expected to use procedural attachment if he wants to see reasoning be performed. This contrasts also with the procedural embedding of knowledge paradigm (Hewitt, 1971) which postulates that every piece of knowledge is embedded in a procedure specifying its usage. However, although we reject the use of procedures for deductive purposes, this somewhat extreme position has its own flaws. Basically there are two reasons why procedures are needed: to deal with infinitary concepts and to allow 'black boxing'.

Consider the integers. Integers are concepts. Because there are infinitely many integers we need an infinite number of concepts. And because each concept needs to be defined, we would get an infinite number of frames! This is of course impossible in a concrete system, so some other way must be found to work with infinite sets of concepts. In particular we need mechanisms, i.e. procedures, that will generate concepts as needed. Concepts like integers will be called *infinitary concepts*. Infinitary concepts are the first reason why we need procedural attachment.

Here is the second reason. In certain situations we want to create a *'black box'* that does a particular task in a way which is nontransparant for the reasoner. A typical example of such a task is arithmetic. In principle it is possible to feed a particular axiomatization of number theory to the reasoner and perform arithmetic operations by performing deductions. It is also possible to create a system based on tables and methods mimicing the way humans do arithmetic. But these approaches are only appropriate when we want to study axiomatic number systems or the way humans perform arithmetic. In most cases we just want the number crunching to be done by a 'black box' as efficiently as possible. The way to create such a black box is to use a programming language as the device to construct boxes with. A particular box is then invoked by associating a procedure with an aspect in a frame. Each time an instantiation of the frame is created, an instance of the box is put at work. Another example where such a black box approach might be useful is in dealing with 'low level' tasks such as vision or sensori-motor behavior. The black box in this case is a channel that returns a piece of information or a mechanism that performs a possibly complex action.

In order to accomodate these capabilities we have extended the XPRT-system so that it is possible to use both procedures and descriptions as constraints on what the answer to a question posed by a certain aspect in a frame can be.

The rest of the paper is organized in four parts. The first part deals with infinitary concepts. Then we turn to procedures. We discuss the notion of a value, the representation of procedures, and the way procedures are evaluated. The third part contains some examples of reasoning based on arithmetic constraints and on constraints over strings and sequences. These examples are taken from the domains of electronic circuits and music. In a final part we discuss the relation to other work.

2. INFINITARY CONCEPTS

Before we turn to the subject-matter of procedural attachment, some more clarification is needed on the notion of an infinitary concept. An infinitary concept is the generalization of an infinite series of concepts. Because we are dealing with a system that is constrained in time and space, and because we cannot know in advance which element of the set will be needed, we need mechanisms generating specializations of an infinitary concept based on a constructive definition. Typical examples of infinitary concepts are numbers and strings, lists, sequences and sets.

It is important to keep in mind that the specializations of an infinitary concept are also concepts. That means that if we say of a certain quantity that it is 5, we really mean that this quantity is an instance of the concept 'the quantity 5'. This is different from ordinary mathematics (and programming) where these objects are considered to be constants. In such a framework to say that a certain quantity is 5 means that this quantity is co-referential with the unique individual 5. Our position is however not unusual from a logical point of view. Indeed since Frege it has been customary to view a number n as a concept expressing the notion of a class of n members.

In order to deal with the problems of reading, printing and internally representing infinitary concepts we will use the mechanisms already available in the programming language in which the reasoning system is implemented (i.e. LISP). Macro's have been created that translate an occurrence of an infinitary concept into a new one that makes reference to the frame of the generic concept. All properties can then be attached to that frame. For example, if [and] are used to represent sequences, then each occurrence such as

```
[1 2 3]
```

will be translated into a new description roughly of the following form:

```
(AND (A SEQUENCE)
     [1 2 3])
```

so that it becomes known to the reasoner that the object described as [1 2 3] is a sequence. All properties of sequences in general are associated with the frame for SEQUENCE and will therefore be inherited by the instances of [1 2 3].

For the rest of the text, the following conventions will be used for infinitary concepts:
(i) Lists are represented with quotes and round-brackets as in '(A B C)
(ii) Numbers are represented in the usual way: 1,2,...
(iii) Sequences are represented with angular brackets as in [1 2 3].
(iv) Strings are represented as quoted atoms as in 'John. The string itself is the atom's p-name.
(v) Sets are represented with curly brackets as in {1 2 3}.
When a user wants to introduce his own class of infinitary concepts, he can introduce appropriate read-macro's and a generic frame for the whole class.

3. PROCEDURAL ATTACHMENT

We now turn to procedures. Two issues need to be considered. How to represent procedures and how to organize their evaluation.

3. 1. REPRESENTING PROCEDURES AND VALUES

It makes sense to use the language that was used to implement the reasoning system, i.e. LISP, itself as the language to represent procedures. A procedure call is represented in LISP by writing the name of the procedure followed by its arguments, as in

```
(+ 1 2)
```

A procedure takes as arguments the values of its actual parameters and returns something as its value. We will allow two kinds of things to be arguments or returned as values:

+ occurrences of infinitary concepts, like
  - numbers
  - strings
  - lists, sequences or sets of occurrences of infinitary concepts
+ lists, sequences or sets of objects in the model of the problem situation.

What do we mean here by an object in the model of the problem situation ?

As mentioned earlier on, the XPRT-system builds up a model of the problem situation it is dealing with. This model consists of objects, descriptions of objects and rules which are working on the expansion of the model. An object is the analogue of an individual in the domain of discourse. A description states the role an object plays in one of the relations which holds in the problem situation. Another way to say that an object is described by a certain description is to say that the object is a referent of that description (not necessarily the only one). In order to identify an object in a particular model, each object has a unique name which is viewed as one of its descriptions.

The necessity of having a list or sequence or set of names of objects as value of another object is best illustrated by an example. Suppose we want to talk about the set of children of a family. Then we will have an object in the model (say CHILDREN-1) which denotes the set of children. If we now want to describe this set by enumeration we need the ability to say something like

```
{JOHN-1, MARY-1, GEORGE-1}
```

where JOHN-1, MARY-1 and GEORGE-1 are the names of the objects which are individuals in the set of children.

Now note that everything which can be a value, i.e. can function as the result of a procedure or as one of its arguments is a description. This fact constitutes the bridge between symbolic reasoning and the execution of procedures. When a procedure is attached to a particular object in the domain of the model, this means the same as saying that the result of this procedure is a description of this object.

In order to talk about *the* value of an object in a particular model of a problem situation, we adopt the convention that an object can have only one value. When an object has more than one value, a contradiction has been reached.

But now there is an ambiguity problem: When we use a certain description it is unclear whether we mean the object denoted by the description or the value of this object. Ambiguities are resolved by adopting additional conventions. The most useful convention seems to be that inside a procedure call a description refers to the value of the object referred to by the description (and there is an error if this description has no unique referent) and outside a procedure call a description refers to the object itself.

So an expression of the form

```
(+ (= the-first-arg) 10)
```

denotes the result of calling the LISP-procedure + with the value of the referent of (= THE-FIRST-ARG) as first argument and the value 10 as second argument. Recall (from Steels,1979) that (= THE-FIRST-ARG) refers to the filler of the FIRST-ARG aspect in the frame where this description occurs. For example, if there would be a frame like

```
(NAME-OF-FRAME
  (WITH FIRST-ARG ...) ....)
```

then the expression (= THE-FIRST-ARG) anywhere in this frame refers to the filler of the first-arg in the instantiation of the frame under consideration.

Here is another example illustrating the procedural conventions. We introduce a frame for SUM which holds between a sum, an addend and an augend:

```
(FRAME SUM
  (WITH SELF
        (+ (= THE-ADDEND) (= THE-AUGEND)))
  (WITH ADDEND
        (- (= THE-SELF) (= THE-AUGEND)))
  (WITH AUGEND
        (- (= THE-SELF) (= THE-ADDEND))))
```

The SELF, ADDEND and AUGEND aspects all have procedures attached to them. For example the procedure attached to the SELF-aspect, in particular

```
(+ (= THE-ADDEND) (= THE-AUGEND)),
```

says that the value of the object which fills the SELF-aspect is described as the result of a procedure. This procedure is the LISP +, which takes two arguments. The first argument is the value of the referent of the description (= THE-ADDEND), i.e. the filler of the addend-slot in the instance of SUM, and the second argument is the value of the referent of the description (= THE-AUGEND), i.e. the filler of the augend-slot in the instance of SUM.

Suppose an instance of SUM has the following fillers:

```
(SUM
  (WITH SELF SUM-1)
  (WITH ADDEND ADDEND-1)
  (WITH AUGEND AUGEND-1))
```

then the procedure attached to the SELF aspect, i.e. SUM-1, would in this case be

```
(+ ADDEND-1 AUGEND-1)
```

So that when the values of ADDEND-1 and AUGEND-1 are 2 and 3, then we know that

the value of SUM-1 is 5.

Although the adopted convention is very natural it fails in cases where we want to refer to the value outside a procedure call or to the object itself inside a call. The second case is necessary in circumstances where we want a list, a set or a sequence of objects in the domain of the model. For example, we might want to talk about a set which contains John, the father of Mary and George.
To resolve this problem we adopt the convention that inside a procedure call we write <o> before a description if we need the object referred to by the description rather than the value of the object and outside a procedure call we write <v> if we need the value of the object of the description rather than the object itself. The o in <o> stands for object, whereas the v in <v> stands for value.

Here is an example where this convention is useful. Suppose we want to talk about telephone-numbers and make a distinction between the object itself and the numerical value of the number. (A telephone-number can consist of letters!) Suppose we have a frame for telephone-number as

```
(FRAME TELEPHONE-NUMBER
 (WITH SELF)
 (WITH NUMBER)
 (WITH OWNER))
```

then we can say that the number is the value of the self of the telephone-number by writing:

```
(FRAME TELEPHONE-NUMBER
 (WITH SELF)
 (WITH NUMBER <v>(= THE-SELF))
 (WITH OWNER))
```

So if we now want to say

```
  34265 IS JOHN'S TELEPHONE-NUMBER
```

or stated in terms of the description language

```
>> 34265 IS (A TELEPHONE-NUMBER (WITH OWNER JOHN))
```

then the NUMBER-aspect of this instance will be filled with the number 34265.

Note that in a system like that of Sussman and Steele (1978) this problem does not occur because there a description always refers to the value of its object.

3. 2. EVALUATION

Ideally we want any procedure to be evaluated as soon as possible. This is a problem however because of the following points:
(i) The values of the arguments to the procedure are not necessarily known at the time we know that the procedure is to be executed. For example, given

```
>> L-1 IS (A SUM
            (WITH ADDEND L-2)
            (WITH AUGEND L-3))
```

then it could be that we do not know the values of L-2 and L-3 at the time this

predication is performed.
(ii) The values of the various objects become known in an arbitrary order. This is especially true because we are operating in a parallel processing context where no decision should be based on the time when something becomes known.

Based on this analysis we propose the following scheme to incorporate procedural attachment into the XPRT-reasoning system. Recall that a model in this system consists of a set of active objects called *experts*. An expert is a special kind of actor (Hewitt, Bishop and Steiger (1973)). Experts communicate with each other and thus perform reasoning behavior. Each of these experts is responsible for one object or individual in the problem situation covered by the model. An expert has a set of descriptions which specify the roles that the object covered by the expert plays in various instantiations. The expert also has a set of rules which wait for descriptions that enter as messages in the expert and will respond in an appropriate way.

We now extend this picture as follows. First of all an expert keeps track of which description in its description set is the value. (It can be that an expert does not yet have a value.) Second an expert also maintains a special set of rules further called procedural rules. Procedural rules are responsible for the procedures that have been predicated for the expert's object. Such a rule can be thought of as a lambda-expression

```
(LAMBDA (X) (procedure ...))
```

The procedural rules wait until the value of the expert in which they are located is known. If so a lambda-conversion is performed with the value of the expert as the value of the lambda-variable. Then the body of the rule is executed. This body will consist either in a new procedural-rule to be sent to an expert or in a value to another expert to be predicated of an object in the model.

When an expert receives a description which is marked as its value, then all waiting procedural-rules have to be executed. When an expert receives a procedural-rule and a value is known, the procedural-rule is executed. When an expert receives a procedural-rule and no value is known yet, the expert adds the rule to its list of procedural rules.

All this means that an expression like

```
(+ ADDEND-1 AUGEND-1)
```

predicated for SUM-1 has to be decomposed in the following way: A procedural rule must be sent to the ADDEND-1 expert. The body of this rule is another procedural-rule to be sent to the AUGEND-1 expert. The body of the latter rule contains a specification to send the result of the evaluation of + to SUM-1.
The following account, albeit unrealistic, will give a clearer idea of what is meant here. Suppose ADDEND-1 receives the following procedural rule:

```
(LAMBDA (X)
        (SEND-PROCEDURAL-RULE
         AUGEND-1
         (LAMBDA (Y) (SEND-DESCRIPTION SUM-1 (+ X Y)))))
```

Suppose ADDEND-1 receives the value 2, then after lambda-conversion the following transmission occurs:

```
(SEND-PROCEDURAL-RULE
 AUGEND-1
 (LAMBDA (Y) (SEND-DESCRIPTION SUM-1 (+ 2 Y ))))))
```

So AUGEND-1 gets the following rule:

```
(LAMBDA (Y) (SEND-DESCRIPTION SUM-1 (+ 2 Y)))
```

Suppose AUGEND-1 has or receives the value 3, then after lambda-conversion the following transmission takes place:

```
(SEND-DESCRIPTION SUM-1 (+ 2 3))
```

After evaluation of (+ 2 3), SUM-1 will be described as having 5 as its value.

4. EXAMPLES

We will now go through several examples. The first example is taken from the domain of electrical engineering and illustrates reasoning with based on arithmetic constraints. The second example illustrates the use of strings and sequences and comes from the domain of musical reasoning.

4. 1. ELECTRONIC NETWORKS

In order to make comparisons easy, we will redo the examples of Sussman and Steele (1978). First we need some frames for arithmetic constraints.

```
(FRAME SUM
  (WITH SELF
        (PLUS (= THE-ADDEND) (= THE-AUGEND)))
  (WITH ADDEND
        (DIFFERENCE (= THE-SELF) (= THE-AUGEND)))
  (WITH AUGEND
        (DIFFERENCE (= THE-SELF) (= THE-ADDEND))))
```

SELF is the name of an aspect that introduces the object (situation, relation, quality,...) of the frame itself.

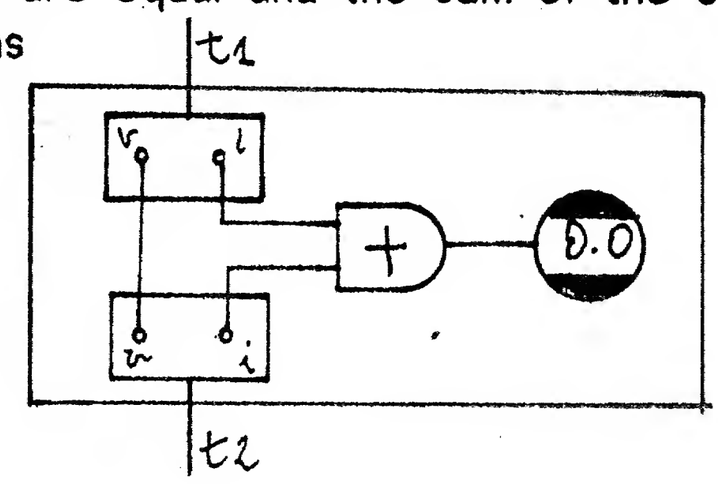
```
(FRAME PRODUCT
  (WITH SELF
        (TIMES (= THE-MULTIPLICAND)(= THE-MULTIPLIER)))
  (WITH MULTIPLICAND
        (QUOTIENT (= THE-SELF)(= THE-MULTIPLIER)))
  (WITH MULTIPLIER
        (QUOTIENT (= THE-SELF) (= THE-MULTIPLICAND)))).
```

PLUS, DIFFERENCE, TIMES, and QUOTIENT are LISP-functions. Observe that whenever sufficient information is available the other aspects of the constraint will be computed.

Next we introduce frames for components of circuits. First comes a frame for a terminal. A terminal has a potential (represented by the voltage-aspect) and a current:

```
(FRAME TERMINAL
  (WITH SELF)
  (WITH VOLTAGE)
  (WITH CURRENT)
  (CRITERIAL (SELF)))
```

We can put two terminals together into a two-terminal node or 2-node. In a 2-node the potentials of the two terminals are equal and the sum of the currents has to be 0.0. Graphically this is represented as

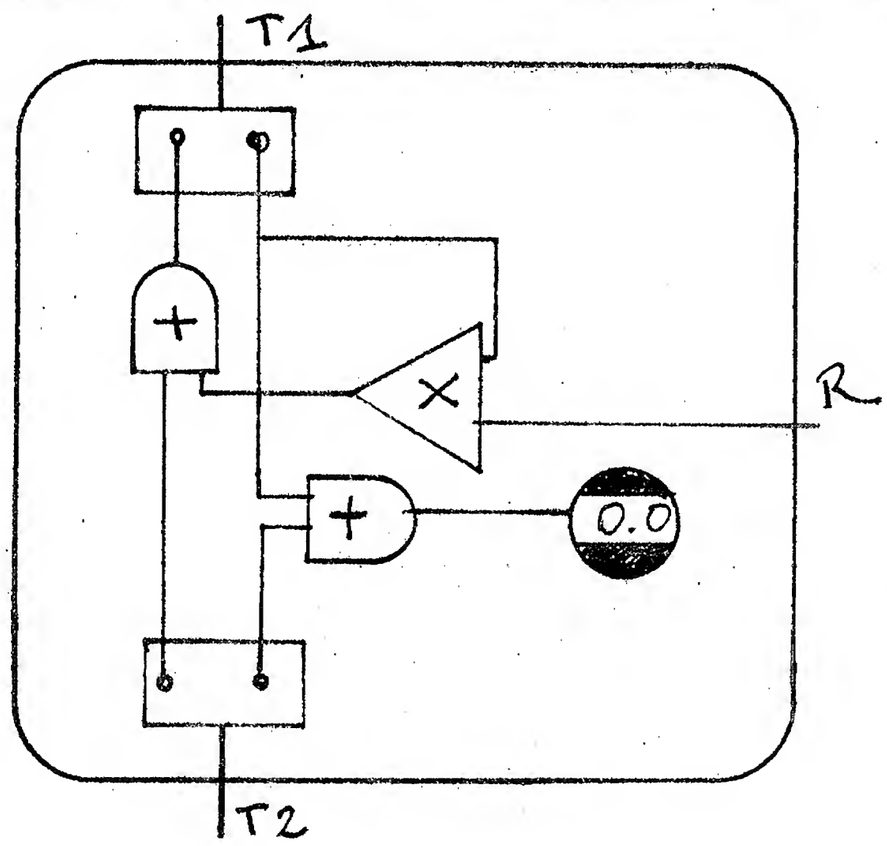


2-node

This can be expressed in descriptions by saying that a 2-node is an object with aspects for two terminals: terminal1 and terminal2. The terminal2 is described as a terminal whose voltage is equal to the voltage of the terminal1 and whose current is the augend of a sum whose result is 0.0 and whose addend is the current of terminal1:

```
(FRAME 2-NODE
       (WITH SELF)
       (WITH TERMINAL1)
       (WITH TERMINAL2
             (A TERMINAL
                (WITH VOLTAGE
                      (THE VOLTAGE OF A TERMINAL
                           (WHICH IS (= THE-TERMINAL1))))
                (WITH CURRENT
                      (THE AUGEND OF A SUM
                           (WHICH IS 0.0)
                           (WITH ADDEND
                                 (THE CURRENT OF A TERMINAL
                                      (WHICH IS (= THE-TERMINAL1)))))))))
       (CRITERIAL (SELF)))
```

A resistor involves two terminals and a resistance which are related via two SUM-constraints and a PRODUCT-constraint as illustrated in the following diagram:

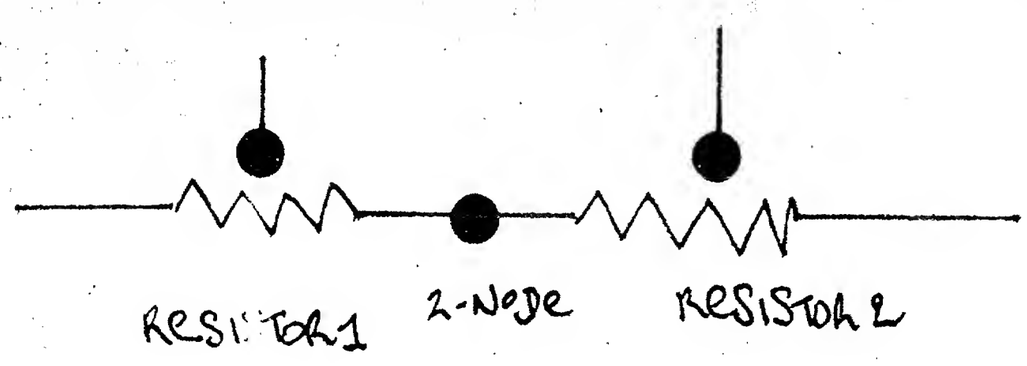


RESISTOR

The resistor-frame itself has aspects for the resistor itself, two terminals called terminal1 and terminal2 and a resistance. The current of the first terminal is described as the augend of a sum whose result is 0.0 and whose addend is the current of the other terminal. The voltage of the first terminal is described as a sum with the voltage of the other terminal as addend and the product of the resistance and the current of the first terminal as augend.

```
(FRAME RESISTOR
       (WITH SELF)
       (WITH TERMINAL2)
       (WITH RESISTANCE)
       (WITH TERMINAL1
             (A TERMINAL
                (WITH CURRENT (= THE-CURRENT-OF-TERMINAL1)
                      (THE AUGEND OF A SUM
                           (WHICH IS 0.0)
                           (WITH ADDEND
                                 (THE CURRENT OF A TERMINAL
                                      (WHICH IS (= THE-TERMINAL2))))))
                (WITH VOLTAGE
                      (A SUM
                         (WITH ADDEND
                               (THE VOLTAGE OF A TERMINAL
                                    (WHICH IS (= THE-TERMINAL2))))
                         (WITH AUGEND
                               (A PRODUCT
                                  (WITH MULTIPLICAND (= THE-RESISTANCE))
                                  (WITH MULTIPLIER
                                        (= THE-CURRENT-OF-TERMINAL1)))))))))
(CRITERIAL (SELF)))
```

We will now look at a simple-circuit which involves two resistors connected by a 2-node as illustrated in the following diagram:



In other words the terminal2 of the first resistor is connected to the terminal1 of the 2-node and the terminal2 of the 2-node is connected to the terminal1 of the second resistor. This information can be represented easily by describing the first resistor of a simple-circuit as a resistor whose terminal2 is the terminal1 of the 2-node and by describing the second resistor as a resistor whose terminal1 is the terminal2 of the 2-node.

```
(FRAME SIMPLE-CIRCUIT
       (WITH SELF)
       (WITH RESISTOR1
             (A RESISTOR
                (WITH TERMINAL2
                      (THE TERMINAL1 OF A 2-NODE
                           (WHICH IS (= THE-2-NODE))))))
       (WITH RESISTOR2
             (A RESISTOR
                (WITH TERMINAL1
                      (THE TERMINAL2 OF A 2-NODE
                           (WHICH IS (= THE-2-NODE))))))
       (WITH 2-NODE)
       (CRITERIAL (SELF)))
```

Here is a dialogue with the current implementation of the reasoner based on these frames. First we create an instantiation for a simple-circuit and call it S-C.

```
>> S-C is (a simple-circuit)
```

One thing we can do is inquire whether S-C is indeed a circuit:

```
>> Is s-c (a simple-circuit) ?
YES
S-C IS (A SIMPLE-CIRCUIT
          (WITH 2-NODE 2-NODE-1)
          (WITH RESISTOR2 RESISTOR2-1)
          (WITH RESISTOR1 RESISTOR1-1))
```

RESISTOR1-1, RESISTOR2-1, etc. are the internal names of the object-experts that have been created for the various components of the circuit. Each of these components is worked out completely as illustrated by the following queries:

```
>> Is resistor1-1 (a resistor) ?
YES
RESISTOR1-1 IS (A RESISTOR
                  (WITH RESISTANCE RESISTANCE-2)
                  (WITH TERMINAL2 TERMINAL1-1)
                  (WITH TERMINAL1 TERMINAL1-2))
>> Is Terminal1-1 (a terminal) ?
YES
TERMINAL1-1 IS (A TERMINAL
                  (WITH VOLTAGE VOLTAGE-1)
                  (WITH CURRENT CURRENT-1))
```

etc;

Now we start supplying values to certain parts of the circuit. First we specify the voltage of the first terminal of the first resistor in the circuit:

```
>> (the voltage of a terminal
       (which is the terminal1 of a resistor
              (which is
                   (the resistor1 of a simple-circuit
                        (which is s-c)))))
   is 10.0
```

Next we specify some of the parts of the second resistor in the circuit:

```
>> (the resistor2 of a simple-circuit
        (which is s-c))
   is
    (a resistor
          (with resistance 9.0)
          (with terminal1 (a terminal (with voltage 3.0)))
          (with terminal2 (a terminal (with voltage 0.0))))
```

At this point sufficient information is available to deduce the values of all the components of the circuit. Let us probe for one

```
>> What is (the resistance of a resistor
                (which is (the resistor1 of a simple-circuit
                              (which is s-c)))) ?
21.0
```

Further justifications will show how this value was obtained.

```
>> Why ?
RESISTANCE-2 IS 21.0
BECAUSE
1.   : CURRENT-4 IS 0.333333332
2.   : AUGEND-1 IS 7.0
3.   : RESISTANCE-2 IS (QUOTIENT AUGEND-1 CURRENT-4)
>> Why ?
CURRENT-4 IS 0.333333332
BECAUSE
1.   : CURRENT-2 IS -0.333333332
2.   : SUM-2 IS 0.0
3.   : CURRENT-4 IS (DIFFERENCE SUM-2 CURRENT-2)
>> Why ?
CURRENT-2 IS -0.333333332
BECAUSE
1.   : CURRENT-1 IS 0.333333332
2.   : SUM-1 IS 0.0
3.   : CURRENT-2 IS (DIFFERENCE SUM-1 CURRENT-1)
>> Why ?
CURRENT-1 IS 0.333333332
BECAUSE
1.   : RESISTANCE-1 IS 9.0
2.   : AUGEND-2 IS 3.0
3.   : CURRENT-1 IS (QUOTIENT AUGEND-2 RESISTANCE-1)
>> Why ?
RESISTANCE-1 IS 9.0
BECAUSE
RESISTANCE-1 IS
(THE RESISTANCE OF A RESISTOR
     (WITH TERMINAL1 TERMINAL1-1)
     (WITH TERMINAL2 TERMINAL2-1))
```

```
>> Why ?
RESISTANCE-1 IS (THE RESISTANCE OF A RESISTOR
                    (WHICH IS RESISTOR2-1)
                    (WITH TERMINAL1 TERMINAL1-1)
                    (WITH TERMINAL2 TERMINAL2-1))
BECAUSE
RESISTOR2-1 IS (A RESISTOR
                   (WITH RESISTANCE 9.0)
                   (WITH TERMINAL1 (A TERMINAL (WITH VOLTAGE 3.0)))
                   (WITH TERMINAL2 (A TERMINAL (WITH VOLTAGE 0.0))))
>> Why ?
RESISTOR2-1 IS (A RESISTOR
                   (WITH RESISTANCE 9.0)
                   (WITH TERMINAL1 (A TERMINAL (WITH VOLTAGE 3.0)))
                   (WITH TERMINAL2 (A TERMINAL (WITH VOLTAGE 0.0))))
BECAUSE YOU TOLD ME SO
```

4. 2. MUSIC

The second example illustrates the use of strings and sequences. The external representation of a sequence makes use of square brackets, as in

```
[1 2 3]
```

This representation is translated internally such that the object which is described in terms of a sequence is now known to be a sequence based on the following frame:

```
(FRAME SEQUENCE
  (WITH SELF)
  (CRITERIAL (SELF)))
```

The value of an object which is described as a sequence is a list-representation containing all the members of the sequence.

The frame for string looks like this:

```
(FRAME STRING
  (WITH SELF (STRING-MAKER (= THE-PARTS)))
  (WITH PARTS)
  (CRITERIAL (SELF)))
```

STRING-MAKER is a LISP-functions which makes a string out of the p-names of a list of atoms. Note that an object that is described as a string has this string as its value.

Let us now illustrate the use of these frames with an example from the domain of music. In particular we consider the problem of constructing the name of a pitch given the name of the octave and the name of the tone.

A pitch has an octave and a tone, so that we can start from the following basic frame:

```
(FRAME PITCH
  (WITH SELF)
  (WITH OCTAVE)
  (WITH TONE)
  (CRITERIAL (SELF) (OCTAVE TONE)))
```

We want a numerical representation for musical objects to perform efficient arithmethic computations on them if necessary. Thus a tone has as value a number ranging from 0 to 11. An octave has a value ranging (in the present discussion) from 0 to 8. Pitch has a value which ranges from 0 to 107. The first thing we will do is establish the constraint between these different values.

```
(FRAME PITCH
   (WITH SELF)
   (WITH OCTAVE
        (AN INTEGER-QUOTIENT
             (WITH DIVIDEND (= THE-SELF))
             (WITH DIVISOR 12)
             (WITH REMAINDER (= THE-TONE))))
   (WITH TONE)
   (CRITERIAL (SELF) (OCTAVE TONE)))
```

We made use here of a frame for integer-quotient which looks like this

```
(FRAME INTEGER-QUOTIENT
       (WITH DIVIDEND
             (+ (* (= THE-SELF) (= THE-DIVISOR))
                (= THE-REMAINDER)))
       (WITH SELF
             (// (= THE-DIVIDEND) (= THE-DIVISOR)))
       (WITH DIVISOR
             (// (- (= THE-DIVIDEND) (= THE-REMAINDER))
                 (= THE-SELF)))
       (WITH REMAINDER
             ( (= THE-DIVIDEND) (= THE-DIVISOR)))
       (CRITERIAL (SELF)))
```

Now we turn to the names of pitches. We will assume that octaves are names and tones have explicit names. (How you derive the names of the tones is a complicated story). So frames for TONE and OCTAVE will look like this

```
(FRAME TONE
  (WITH SELF)
  (WITH NAME))
```

and

```
(FRAME OCTAVE
  (WITH SELF))
```

The names of tones and octaves are strings. Thus the tone A can be defined as follows:

```
(FRAME A-TONE
  (WITH SELF
        (AND (A TONE (WITH NAME 'A))
             0)))
```

Note that 0 is the value of the A-tone.

Here is then the frame for pitch again, this time with a constraint for the name :

```
(FRAME PITCH
   (WITH SELF)
   (WITH NAME
         (A STRING
            (WITH PARTS
                  [(THE NAME OF A TONE
                        (WHICH IS (= THE-TONE)))
                  '-
                  (= THE-OCTAVE)])))
   (WITH OCTAVE
        (AN INTEGER-QUOTIENT
             (WITH DIVIDEND (= THE-SELF))
             (WITH DIVISOR 12)
             (WITH REMAINDER (= THE-TONE))))
   (WITH TONE)
   (CRITERIAL (SELF)(OCTAVE TONE)))
```

The constraint makes use of the STRING-frame introduced earlier on. The parts of this frame form a sequence, which will be composed together in a single string.

Let us now experiment with these frames.

```
>> (P-1 is (a pitch
               (with tone (a tone (with name 'c)))
               (with octave 2)))
>> What is (the name of a pitch
                 (which is p-1))
C-2
>> Why ?
NAME-1 IS C-2
BECAUSE
1.   : PARTS-1 IS (C - 2.)
2.   : NAME-1 IS (STRING-MAKER PARTS-1)
>> Why ?
PARTS-1 IS (C - 2.)
BECAUSE
1.   : OCTAVE-1 IS 2.
2.   : NAME-2 IS C
3.   : PARTS-1 IS (SEQUENCE-FUNCTION (THE NAME OF A TONE (WHICH IS TONE-1))
                                    - OCTAVE-1))
```

Sequence function creates a sequence by taking the values of each description in the sequence.

We now illustrate the computation of the numerical representation of a pitch, by supplying the missing piece of information: the value of the tone C:

```
>> (the tone of a pitch
        (which is p-1))
   is 3
```

The value of the pitch has now been computed:

```
>> What is p-1 ?
27.
>> Why ?
P-1 IS 27.
BECAUSE
1.   : TONE-1 IS 3.
2.   : DIVIDEND-1 IS 12.
3.   : OCTAVE-1 IS 2.
4.   : P-1 IS (PLUS (TIMES OCTAVE-1 DIVIDEND-1) TONE-1))
```

5. CONCLUSION

In this paper we have extended the XPRT-system in order to deal with procedural constraints on what things can be the fillers of the slots in a frame. It turned out that incorporating this new capability was a fairly straightforward matter, given the way the XPRT-system was designed and implemented.
In a second part of the paper a number of examples from the domains of electronic circuits, and music were discussed. These examples illustrate arithmetic constraints and processing with strings and sequences.

The efforts documented in this paper are part of ongoing research in message passing systems and description systems (in particular frame-systems). Our frames have become similar to the classes of Simula (Dahl, et.al. 1973) and Smalltalk (Kay,1976) in the sense that information is organized in units who have components and procedures attached to these components and who are related to each other via inheritance relations. Our system is on the other hand of a higher level because the user no longer has to specify explicitly the message passing that should occur. Instead he has the ability to specify constraints which translate (in a dynamic way) into collections of messages. In this sense our efforts are similar to the ones of Borning (1979). The same effect has also been obtained in the constraint system of Sussman and Steele (1978). This system is even more powerful than ours because it has retraction capacities which we have not yet implemented.

On the other hand the system discussed in this paper goes beyond the constraint systems of Borning and Sussman and Steele because we are able to do symbolic reasoning. For example, in the constraint system of Sussman and Steele (1978) it is not possible to ask a conditional question like 'Is this object a resistor?'. It is therefore not possible to perform qualitative reasoning.

A system similar in capacity to the one discussed here is proposed in Hewitt, Attardi and Lieberman (1979). The major difference lies in the fact that this system is not frame-based and is not internally organized around objects but around descriptions.

Finally it must be stated that this work forms part of a continuing development of frame-based systems. In comparison to earlier systems like KRL (Bobrow and Winograd, 1977) and FRL (Roberts and Goldstein,1977), we see that although the approach to the organization of knowledge is the same, the methods for activating it are radically different. KRL relies on a matching mechanism, whereas FRL expects the user to attach procedures whenever he wants to see reasoning be performed.

6. ACKNOWLEDGEMENT

I am indebted to many members of the MIT Artificial Intelligence Laboratory for their help in shaping the ideas expressed in this paper and in getting them on paper. In addition, W. Martin, C. Hewitt, R. Duffey and G. Attardi have contributed to this particular paper.

7. REFERENCES

BOBROW, D. AND T. WINOGRAD (1977) An overview of KRL - A Knowledge Representation Language. Cognitive Science, 1.1.

BORNING, A. (1979) THINGLAB -- An object-oriented system for building simulations using constraints. STANFORD AI-Lab. Ph.D. thesis

DAHL, O. and Nygaard, K. (1968) Class and Subclass Declarations. In Simulation Programming Languages. J.N. Buxton (ed.) North Holland.

GOLDSTEIN, I. and B. ROBERTS (1977) The FRL-manual. MIT-AI memo 409.

HEWITT, C. (1971) Procedural Embedding of knowledge in PLANNER. IJCAI-71. London.

HEWITT, C., P. BISHOP AND R. STEIGER (1973) A Universal Actor Formalism for Artificial Intelligence. IJCAI-73. Stanford University.

HEWITT, C., G. ATTARDI AND H. LIEBERMAN (1979) Specifying and Proving Properties of Guardians for Distributed Systems. in: Semantics of Concurrent Computation. Springer-Verlag, New York.

STEELE, G.L. AND G.J. SUSSMAN (1978) Constraints. MIT-AI memo 502.

STEELS, L. (1979) Reasoning Modeled as a Society of Communicating Experts. MIT-AI TR 542.